

Object-Oriented Programming

Scientific Programming with Python

Christian Elsasser

University of Zurich
Faculty of Science

13 July 2026

Contributors: Niko Wilbert, Roman Gredig, Christian Elsasser, Andreas Weiden, Jonas Eschle, Nicola Chiapolini



This work is licensed under the [Creative Commons Attribution-ShareAlike 4.0 License](https://creativecommons.org/licenses/by-sa/4.0/).

Outline

What is OOP?

Fundamental Principles of OOP

Specialities in Python

Science Examples

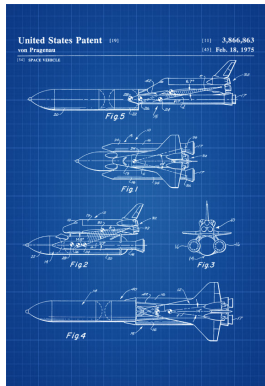
Design Patterns

Setting the scene

Object-oriented programming is a **programming paradigm**.

- ▶ Imperative programming
 - ▶ **Object-oriented**
 - ▶ Procedural
- ▶ Declarative programming
 - ▶ Functional
 - ▶ Logic

What is Object-Oriented Programming?



Segment programs into instances of different object classes:

- ▶ **Instance variables** to describe the state of the object
- ▶ **Methods** to model the behaviour of the object

A **class** definition is like a **blueprint**. Programs create instances and execute their methods.

Why might OOP be a good idea?

DRY (Don't repeat yourself):

Create class functionality once, use repeatedly across programs.

Inheritance allows easy creation of new classes by extending existing ones.

KIS (Keep it simple):

Split functionality into **basic building blocks** and **algorithms invoking them**. This creates a natural code structure.

Complex problems require more than single instruction sequences for maintainable code.

Example of a class

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color

    def make_sound(self):
        print("Wuff!")

# create an instance 'snoopy' of the class Dog
snoopy = Dog()

# first argument (self) is this Dog instance
snoopy.make_sound()

# change snoopy's color
snoopy.color = "yellow"
```

- ▶ Start with `class` keyword.
- ▶ Methods: functions in class scope with `self` as first argument.
- ▶ `__init__`: called when creating new instances.
- ▶ Data attributes are defined in `__init__`.

Fundamental Principles of OOP (I)

Encapsulation

- ▶ Expose only **necessary parts** (public interface).
- ▶ Hide implementation details for abstraction.
- ▶ Break large problems into understandable parts.

In Python:

- ▶ No explicit private/public declarations.
- ▶ Convention: private parts start with `_`.
- ▶ Uses **documentation** and **conventions** instead of enforcement.

Example of Encapsulation

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._mood = 5

    def _change_mood(self, change):
        self._mood += change
        self.make_sound()

    def make_sound(self):
        if self._mood < 0:
            print("Grrrrr!")
        else:
            print("Wuff!")

    def pat(self):
        self._change_mood(1)

    def beat(self):
        self._change_mood(-2)
```

- ▶ Use pat and beat methods to change mood.
- ▶ Don't access _mood or _change_mood directly.

Fundamental Principles of OOP (II)

Inheritance

- ▶ Create subclasses that **inherit/extend parent classes**.
- ▶ Override methods, add specialized behavior.

In **Python**:

- ▶ Inherit from one/multiple classes (multiple not recommended).
- ▶ Access parent methods via `super`.
- ▶ All classes derive from `object` (implicit).

Example of Inheritance

```
class Mammal:
    def __init__(self, color="grey"):
        self.color = color
        self._mood = 5

    def _change_mood(self, change):
        self._mood += change
        self.make_sound()

    def make_sound(self):
        raise NotImplementedError

    def pat(self):
        self._change_mood(1)

    def beat(self):
        self._change_mood(-2)
```

in mammal.py

```
from mammal import Mammal

class Dog(Mammal):
    def __init__(self, color="brown"):
        super().__init__(color)

    def make_sound(self):
        if self._mood < 0:
            print("Grrrrr!")
        else:
            print("Wuff!")
```

- ▶ `super().__init__(color)`: calls parent constructor.
- ▶ `super`: accesses parent class methods.
- ▶ Used when extending parent methods.

Fundamental Principles of OOP (III)

Polymorphism

- ▶ **Treat subclasses like parent class**, execute specialized behavior.
- ▶ *Example:* All mammals make sounds; dogs bark.

In Python:

- ▶ Python is a **dynamically typed language**: the type (class) of a variable is only known when the code runs.
- ▶ **Duck Typing**: No need to know class of object if it provides the required methods: “If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.” (and we treat it as a duck)
- ▶ Type checking can be performed via the `isinstance` function, but generally prefer duck typing and polymorphism.

Example of Polymorphism

```
from animals import Dog, Cat, Bear

def caress(mammal, number_of_pats):
    if isinstance(mammal, Bear):
        raise TypeError("Bad Idea!")
    for _ in range(number_of_pats):
        mammal.pat()

d, c, b = Dog(), Cat(), Bear()
caress(d, 3) # "Wuff!" (3x)
caress(c, 3) # "Purr!" (3x)
caress(b, 3) # raises TypeError
```

- ▶ caress works for all objects having a method pat
- ▶ Special behaviour for bears:
use `isinstance(mammal, Bear)` to check if mammal is a bear.
- ▶ Dynamic typing makes function overloading like in other languages impossible!

Python Specialities – Magic Methods

```
class Dog:
    def __init__(self, name, color="brown"):
        self.name = name
        self.color = color
        self._mood = 5

    def __repr__(self):
        return f"{self.name}: {self.color} dog"

snowy = Dog("snowy", "white")
print(snowy) # snowy: white dog
```

- ▶ Magic methods (full list [here](#)) start and end with two underscores (“dunder”).
- ▶ They customise standard Python behavior (e.g. string representation or operator definition).

Python Specialities – Property

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._mood = 5

    def _get_mood(self):
        if self._mood < 0:
            return "angry"
        else:
            return "happy"

    mood = property(_get_mood)

# create an instance 'snowy' of the class Dog
snowy = Dog("white")
print("Snowy is", snowy.mood)
```

- ▶ `property()` has upto four arguments:
 1. Getter
 2. Setter
 3. Deleter
 4. Documentation string
- ▶ Access calculated values like data attributes.
- ▶ Create read-only attributes.
- ▶ Preprocess assigned values (see later).

Python Specialities – Property

```
class Dog:
    def __init__(self, color="brown"):
        self.color = color
        self._mood = 5

    @property
    def mood(self):
        if self._mood < 0:
            return "angry"
        else:
            return "happy"

# create an instance 'snowy' of the class Dog
snowy = Dog("white")
print("Snowy is", snowy.mood)
```

- ▶ `property()` has upto four arguments:
 1. Getter
 2. Setter
 3. Deleter
 4. Documentation string
- ▶ Access calculated values like data attributes.
- ▶ Create read-only attributes.
- ▶ Preprocess assigned values (see later).

Python Specialities – Classmethods

```
class Dog:
    def __init__(self, name, color="brown"):
        self.name = name
        self.color = color

    @classmethod
    def from_string(cls, s):
        name, *color = s.split(",")
        if not color or type(color) != str:
            return cls(name)
        return cls(name, color)

snowy = Dog.from_string("snowy,white")
```

- ▶ A classmethod takes as its first argument a class instead of an instance of the class. It is therefore called `cls` instead of `self`.
- ▶ One usecase is to write multiple constructors for a class, e.g.:
 - ▶ Default `__init__`.
 - ▶ From serialized string.
 - ▶ From database/file.
- ▶ Other usecases
 - ▶ Keep track/update existing objects (changes class variable).
 - ▶ Instance-independent methods
 - ▶ ...

Python Specialities – Class attributes

```
class Dog:
    breed = "dog"
    all_ = set()

    def __init__(self, name, color="brown"):
        self.name = name
        self.color = color
        Dog.all_.add(self)

    def __repr__(self):
        return f"{self.name}: {self.color} {self.breed}"

Dog("snowy", "white")
balto = Dog("balto")
balto.breed = "husky"
print(Dog.all_) # {snowy: white dog, balto: brown husky}
```

- ▶ A class can also have attributes that are shared among all its objects.
- ▶ If the attribute is modified, all objects will see this ("class global").
- ▶ **Pitfall assignment:** Assigning to an instance (`balto.breed = "husky"`), creates a new instance attribute, hiding the class one. You need the class to modify the class attribute (`type(balto).breed = "canis"`)

Advanced OOP Techniques

There are many advanced techniques that we didn't cover:

- ▶ **Multiple inheritance:** Deriving from multiple classes; it can create a real mess. Need to understand the Method Resolution Order (MRO) to understand `super`.
 - ▶ **Monkey patching:** Modify classes and objects at runtime, *e.g.* overwrite or add methods.
 - ▶ **Abstract Base Classes:** Enforce that derived classes implement particular methods from the base class.
 - ▶ **Metaclasses:** (derived from `type`), their instances are classes.
-
- ▶ Great way to overcomplicate when feeling clever.
 - ▶ Avoid these - you'll likely regret it (KIS).

Science Examples – Vector

```
class Vector3D:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __add__(self, other):
        return type(self)(self.x + other.x,
                           self.y + other.y,
                           self.z + other.z)

    @property
    def length(self):
        return (self.x**2+self.y**2
                +self.z**2)**0.5

    @length.setter
    def length(self, length):
        scale = length/self.length
        self.x *= scale; self.y *= scale; self.z *= scale

# decorators could be replaced by `length = property(...)`
# but functions would need distinguishable names
```

```
from vector import Vector3D

v1 = Vector3D(0, 1, 2)
v2 = Vector3D(1,-3, 0)
v3 = v1 + v2
print(v3.length) # 3.0
v3.length = 6
print(v3.x, v3.y, v3.z)
```

- ▶ Custom type with optimized behavior.
- ▶ Custom functionality.
- ▶ `type(self)` in `__add__`: simplifies inheritance.
- ▶ `@length.setter`: marks property setter.

Science Examples – Dataset

```
import numpy as np

class Dataset:
    mandatory_metadata = ["label", "color", "marker"]
    def __init__(self, datafile, **metadata):
        for key in self.mandatory_metadata:
            if key not in metadata:
                raise KeyError("Missing metadata", key)
        self.metadata = metadata
        self.data = np.loadtxt(datafile, delimiter=",")
        self.validate()

    def validate(self):
        if self.data.shape != (4, 10):
            raise ValueError("Bad shape of data, has to be (4, 10).")

    @property
    def label(self):
        return self.metadata["label"]

    def peak_row(self):
        return self.data.max(axis=1).argmax()
```

```
from dataset import Dataset

ds = Dataset("data_0.csv",
             label="calibration",
             color="r",
             marker="+")
print(ds.label)
```

- ▶ Store metadata with data.
- ▶ Data validation on load.
- ▶ Calculate derived quantities.

Science Examples – Sensors

```
from urllib.request import urlopen

class Sensor:
    def __init__(self, offset=0, scale_factor=1):
        self.offset = offset
        self.scale = scale_factor

    def get_value(self):
        return (self._get_raw() + self.offset) * self.scale

    def _get_raw(self):
        raise NotImplementedError

class WebSensor(Sensor):
    def __init__(self, url, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._url = url

    def _get_raw(self):
        res = urlopen(self._url)
        return float(res.read())
```

```
from sensors import WebSensor

sensor = WebSensor(
    "https://crbn.ch/sensor", 273
)
print(sensor.get_value())
```

- Combine configuration with functionality.
- Support different sensor access methods.

Science Examples – Value with Uncertainty

```
class UncertVal:
    def __init__(self, value, uncertainty=0):
        self.val = value
        self.std = uncertainty

    def __str__(self):
        return f"{self.val} +/- {self.std}"

    def add(self, other, corr=0):
        variance = (self.std ** 2 + other.std ** 2
                    + 2 * self.std * other.std * corr)
        return type(self)(self.val + other.val,
                          variance ** 0.5)

    def __add__(self, other):
        return self.add(other)
```

```
from uncertval import UncertVal

a = UncertVal(2, 0.3)
b = UncertVal(3, 0.4)
print(a + b) # 5 +/- 0.5
```

- ▶ Group related values.
- ▶ Useful string representation.
- ▶ Operators respect value relationships.

Object-Oriented Design Principles and Patterns

How to do Object-Oriented Design right:

- ▶ **Three strikes and you refactor:** Third repetition → create class/function.
- ▶ Sketch with **pen and paper**.
- ▶ Be pragmatic, not perfectionist about real-world correspondence.
- ▶ **Testability:** good design criterion.

How design principles can help:

- ▶ Design principles tell you in an abstract way what a good design should look like (most come down to loose coupling).
- ▶ Design Patterns are concrete solutions for reoccurring problems.

Some Design Principles

Scope of classes:

- ▶ **One class = one single clearly defined responsibility.**
- ▶ **Favor composition over inheritance.**
Inheritance is not primarily intended for code reuse, its main selling point is polymorphism. “Do I want to use these subclasses interchangeably?”
- ▶ Separate **varying aspects** from **stable ones**.
Open for extension, closed for modification.

How to design (programming) interfaces:

- ▶ **Principle of least knowledge.**
Each unit should have only limited knowledge about other units. Only talk to your immediate friends.
- ▶ Minimize the *surface area* of the interface.
- ▶ **Program to an interface**, not an implementation. Do not depend upon concrete classes.

Design Patterns

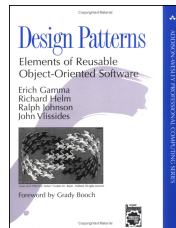
Purpose & background:

- ▶ Idea of concrete design approach for recurring problems.
- ▶ Closely related to the rise of the traditional OOP languages C++ and Java.
- ▶ More important for compiled languages (Open-Closed principle stricter!) and those with stronger enforcement of encapsulation.

Examples:

- ▶ **Decorator pattern**
- ▶ **Strategy pattern**
- ▶ **Factory pattern**
- ▶ ...

Comprehensive list [here](#).



Decorator Pattern



Decorator Pattern – Motivation

Challenge:

- ▶ How to modify the behaviour of an individual object ...
- ▶ ... and allowing for multiple modifications.

Example: Implement a range of products of a coffee house chain

But what about the beloved add-ons?

(Do not confuse the decorator pattern with function decorators!)

```
class Beverage:
    # imagine some attributes like
    # temperature, amount left,...
    _name = "beverage"
    _cost = 0.00

    def __str__(self):
        return self._name

    @property
    def cost(self):
        return self._cost

class Coffee(Beverage):
    _name = "coffee"
    _cost = 3.00

class Tea(Beverage):
    _name = "tea"
    ...
```

Decorator Pattern – First try

Solution:

- Implementation via subclasses

Issue: Number of subclasses explodes to allow for multiple modifications (*e.g.* CoffeeWithMilkAndSugar).

```
class Coffee(Beverage):  
    _name = "coffee"  
    _cost = 3.00  
  
class CoffeeWithSugar(Coffee):  
    _name = "coffee with sugar"  
    _cost = 3.20  
  
class CoffeeWithMilk(Coffee):  
    _name = "coffee with milk"  
    ...
```

Decorator Pattern – Second try

Solution:

- Implementation with switches

Issue: No additional add-ons implementable without changing the class (violation of the open-close principle!).

```
class Beverage:
    _name = "beverage"
    _cost = 0.00

    def __init__(self, milk=False, sugar=False):
        self._milk = milk
        self._sugar = sugar

    def __str__(self):
        desc = self._name
        if self._milk:
            desc += ", with milk"
        if self._sugar:
            desc += ", with sugar"
        return desc

    @property
    def cost(self):
        cost = self._cost
        if self._milk:
            cost += 0.30
        if self._sugar:
            cost += 0.20
        return cost
```

Decorator Pattern – Implementation

Solution:

- ▶ Create a class that wraps a beverage and behaves like a beverage itself. (i.e. implements the beverage interface)
- ▶ Possibility to create a chain of decorators.
- ▶ Composition solves the problem.
- ▶ Downside: Need to implement all functions of beverage even if they do not need to be changed.

```
class Sugar:
    def __init__(self, beverage):
        self.base = beverage

    def __str__(self):
        return f"{self.base}, with sugar"

    @property
    def cost(self):
        return self.base.cost + 0.20

coffee_with_sugar = Sugar(Coffee())
```


Strategy Pattern – Motivation (I)

Let's implement a duck ...

```
class Duck:
    def __init__(self):
        # stateless class for simplicity
        pass

    def quack(self):
        print("Quack!")

    def display(self):
        print("Boring looking duck.")

    def take_off(self):
        print("Run fast, flap wings.")

    def fly_to(self, destination):
        print("Fly to", destination)

    def land(self):
        print("Extend legs, touch down.")
```

Strategy Pattern – Motivation (II)

... and different types of ducks!

Oh, no! The rubber duck should not fly! We need to overwrite all the methods about flying.

- ▶ What if we want to introduce a DecoyDuck as well?
- ▶ What if a normal duck suffers a broken wing?

⇒ It makes more sense to abstract the flying behaviour.

```
class RedheadDuck(Duck):  
    def display(self):  
        print("Duck with a read head.")  
  
class RubberDuck(Duck):  
    def quack(self):  
        print("Squeak!")  
  
    def display(self):  
        print("Small yellow rubber duck.")
```

Strategy Pattern – Implementation (I)

- ▶ Create a class to describe the flying behaviour (flying strategy)...
- ▶ ...give Duck an instance of it ...
- ▶ ...and handle all the flying stuff via this instance

```
class FlyingBehavior:
    def take_off(self):
        print("Run fast, flap wings.")
    def fly_to(self, destination):
        print("Fly to", destination)
    def land(self):
        print("Extend legs, touch down.")

class Duck:
    def __init__(self):
        self.flying_behavior = FlyingBehavior()
    def take_off(self):
        self.flying_behavior.take_off()
    def fly_to(self, destination):
        self.flying_behavior.fly_to(destination)
    def land(self):
        self.flying_behavior.land()
    # display, quack as before...
```

Strategy Pattern – Implementation (II)

- ▶ Other example of composition over inheritance.
- ▶ Encapsulation of function implementation in the strategy object.
- ▶ Useful pattern to *e.g.* define optimisation algorithm at runtime.

```
class NonFlyingBehavior(FlyingBehavior):  
    def take_off(self):  
        print("It's not working :-(")  
    def fly_to(self, destination):  
        raise Exception("I'm not flying.")  
    def land(self):  
        print("That won't be necessary.")  
  
class RubberDuck(Duck):  
    def __init__(self):  
        self.flying_behavior = NonFlyingBehavior()  
    def quack(self):  
        print("Squeak!")  
    def display(self):  
        print("Small yellow rubber duck.")  
  
class DecoyDuck(Duck):  
    def __init__(self):  
        self.flying_behavior = NonFlyingBehavior()  
    # different implementation for display/quack
```

Take-aways

- ▶ Object-oriented programming offers a powerful paradigm to structure your code.
- ▶ Inheritance, design principles and patterns allow to avoid repetitions (DRY).
- ▶ But do not overcomplicate things and always ask yourself if applying a particular functionality makes sense in the given context!

